by Michael Griebling

Hardware Synthesis Using VHDL (Part 1)

VHDL Implementation

Michael presents an XC3S200-based timing system that includes a date/time function, an alarm, and a stopwatch. He used VHDL to get the job done.

Learning something new is never easy. I remember sitting in a brightly lit room surrounded by slackened faces as a presenter droned on and on about VHSIC high-definition language (VHDL). The presenter was an expert in the field who had written a book on the subject. Although he was in demand as a lecturer, his in-depth presentation on VHDL left most of us bored and wondering how to apply what he was teaching.

I'll try to avoid putting you to sleep. My approach is simple: I'll use real-world examples. In this article I'll describe practical aspects of VHDL such as counters, decoders, ROM, state machines, multiplexers, latches, test benches, and much more using a sophisticated stopwatch application (see Figure 1). The application I'll introduce runs on a Xilinx Spartan-3 evaluation board (see Photo 1). Test benches use some constructs that can't be synthesized because they are required to produce a stimulus for testing. I'll warn you about anything that shouldn't be used in a real design. The code constructs are generic VHDL code, which you can use with any synthesis tool. I'll emphasize VHDL code rather than explain how to use a particular tool because most vendors offer extensive tutorials.

WHAT IS VHDL?

VHDL was developed by the U.S. Department of Defense as a common language for describing hardware designs at the system, component, and gate levels. Its main purpose was to enable the exchange of design data in a common format between different contractors.

VHDL is ideal for producing design descriptions that can be translated or synthesized into real hardware implementations. System-C proponents don't want this known, but VHDL is an excellent system modeling language that can describe hardware components at either a functional or gate level.

WHY VHDL?

VHDL is an IEEE standard (Std-1076) supported by all tool vendors. Professionals use it for describing complex ASICs, producing video-processing FPGAs, generating digital signal-processing applications, and much more. By learning VHDL, you will have acquired a skill that's in high demand.

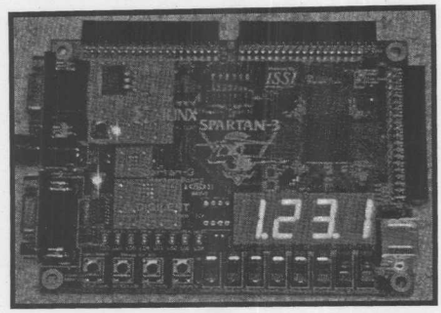


Photo 1—*Take a look at the Xilinx Spartan-3 starter kit with the watch application running in Stopwatch mode. The seven-segment display is located in the lower right corner. The push buttons on the lower left control the stopwatch. An XC3S200 FPGA positioned above the display runs the watch application.*

VHDL is extremely readable with a syntactic structure that's similar to Visual Basic and Pascal. It has a powerful, expressive syntax that can define new data types and inherently support team-based programming. One company I know has dozens of VHDL designers working in numerous locations on one design.

Tom Cantrell recently wrote that the future is bright for FPGAs, which will play a large role in mainstream applications ("More Flash, Less Cash," *Circuit Cellar*, 178, May 2005). I agree with Tom, but I'll go further and predict that VHDL will become the premier technology used to define FPGA content either as output from design tools or with direct programming. In combination with VHDL, FPGAs provide a low-cost approach to defining complex hardware designs that were inconceivable only a few decades ago. Perhaps most importantly, using VHDL to define hardware is fun.

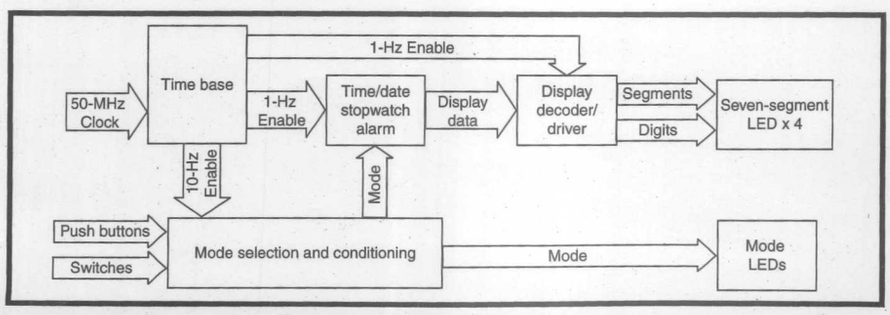


Figure 1—*The watch project consists of a time/date function, a second time zone, an alarm, and a stopwatch with lap timing. A time base generates ENABLE signals to run the timing functions, display the multiplexer, and debounce the circuitry.*

MY KIND OF TYPE

Data types are to VHDL what colors are to a painter. In this respect, VHDL has almost an infinite palette. Table 1 lists all predefined and standardized (Std-1164) data types along with brief descriptions. The only predefined data type I'll cover in this article is something called a natural number. This type represents an integer with values from zero to the largest representable integer (usually a 32-bit value). Later, I'll define some of my own types when I explain how to implement a state machine.

VHDL Type	Built in	Can be synthesized	Values
std_logic	IEEE	Yes	U, X, 0, 1, Z, W, L, H, –
std_logic_vector	IEEE	Yes	Array (natural range <>) of std_logic
boolean	Yes	Yes	True or false
integer	Yes	Yes	32 or 64 bits
natural	Yes	Yes	Integers ≥ 0
positive	Yes	Yes	Integers > 0
bit	Yes	Yes	0, 1
bit_vector(natural)	Yes	Yes	Array (natural range <>) of bits
character	Yes	No	7-bit ASCII
string(positive)	Yes	No	Array (natural range <>) of characters
text	Yes	No	File of string
time	Yes	No	hr, min, sec, ms, us, ns, ps, fs
delay_length	Yes	No	Time ≥ 0

Table 1—*Take a look at some of the data types defined in the Std-1164 and Std-1076 IEEE standards. The most commonly used are the std_logic, std_logic_vector, and natural types.*

Although VHDL includes a bit type with values of 0 and 1, this type usually isn't used for synthesis because it doesn't accurately describe the real-world environment. For example, how would you represent a high-impedance state with the bit type?

To address this problem, the IEEE introduced a new type called `std_logic`, which has values of `U` (uninitialized), `X` (unknown), `W` (weak X), `Z` (high impedance), `H/L` (weak 1/0), `-` (don't care), and, of course, `1` and `0`. The power of VHDL allows the definition of a data type to support any user-defined multi-valued logic using enumerations. Of course, to ensure VHDL portability, it's highly recommended to stick with the `std_logic` type as defined in the Std-1164 package.

An additional type from the IEEE Std-1164 package is called the `std_logic_vector`, which consists of a one-dimensional array of `std_logic` elements. The `std_logic_vector` is used to represent data buses, addresses, and any other aggregate of bits. Only extremely specific `std_logic` values are supported for synthesis purposes: `0`, `1`, `Z`, and `-`. The main reason for this limitation is that the FPGA devices typically contain only gates that respond to 0s and 1s and tristateable buffers using the `Z` value.

Later, I'll describe how to create a tristate buffer. I'll also explain how the `-` value is used when simplifying logic. Keep in mind that when creating test benches or device models, all

of the values of the std_logic type can be used to simulate pull-up and pull-down resistors, undefined states, and logic contention.

OPERATORS

VHDL supports several overloaded operators: logical (and, or, nand, nor, xor, and not, which are applicable to Boolean or bit operations); shift (sll, srl, sra, rll, and rrl, which are only available in some libraries); relational (=, /=, <, <=, >, and >=), arithmetic (+, -, ×, /, mod, rem, **, and abs); and concatenate (&, e.g., two arrays are joined to make a single array).

Any package that introduces a new data type will also overload these operators to enable the manipulation of the new data type and perform conversions to and from existing types and the new data type.

PACKAGES & MORE

I've already cleverly sneaked in the term "package" without really introducing what it means. A package is an encapsulation mechanism used to group related types, constants, and functions or subprograms. Packages are frequently precompiled when the originator prefers not to disclose the source code, but they also can be provided in the form of source code.

The IEEE Std-1164 package is an example of a package supplied with complete source code. I won't describe how packages are created, but I'll refer to functions and types defined in some of the IEEE standard packages. In a real-world project, a package would typically contain the common type and constant declarations and functions required in multiple components. For the purposes of this article, however, the IEEE standard packages will suffice.

Each VHDL component consists of a single entity and one or more architectures. In hardware terms, an entity defines an IC's pins, while the architecture describes the IC's implementation or model. The simplest sort of architecture contains only instantiated components and the wiring interconnects that join the component pins to each other. More complex architectures define detailed functional elements such as counters, registers, state machines, etc.

GETTING STARTED

Listing 1 shows a VHDL implementation of a decade counter somewhat similar to the venerable 74LS90. The first line introduces the standard IEEE library. The next three lines (beginning with use) make visible specific packages contained in this standard: std_logic_1164, std_logic_arith, and std_logic_unsigned. The .ALL postfix indicates that all of the package's contents are made visible within this context. It's also possible to qualify the export of a specific package declaration using this same dot notation.

COUNTER ENTITY

The counter pins are declared in the entity section of the listing near the top. Five signals are defined: clk, enable, reset, q, and carry. Like pins on an IC, signals can have three basic modes (or directions): in, out, or inout.

in mode signals can be read only. out mode signals can be written only. inout signals can be read or written. buffer mode, a lesser-known mode that can be read or written to, is used primarily for carrying information out of a module. I don't recommend using buffer mode, however, because many synthesis tools have trouble with it.

With the exception of q, all other signals are single bits of the std_logic type. The q signal is your first encounter with a bus or std_logic_vector. Typically, bused signal vectors are defined with a bit ordering of n downto 0, which means that the leftmost bit n is also the most significant bit. The opposite bit ordering (from 0 to n) is also possible, but it isn't used too often. The q bus has a width of 4 bits.

COUNTER ARCHITECTURE

In the architecture block, the behavioral name distinguishes this architecture from the different implementations. Because my only focus is on producing real hardware, I won't cover multiple architectures.

The div10_ex architecture name must be the same name as the entity declaration name and tie this architecture to the entity. The architecture's body is contained in a begin/end block with the architecture's name echoed after the end.

Listing 1—*This typical VHDL decade counter/divider implementation features an asynchronous reset, a clock enable, and a carry output.*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity div10_ex is
      port ( clk   : in  std_logic;
           enable  : in  std_logic;
           reset   : in  std_logic;
           q       : out std_logic_vector(3 downto 0);
           carry   : out std_logic);
end div10_ex;
architecture Behavioral of div10_ex is
  signal ctr : std_logic_vector(3 downto 0);
begin
  q <= ctr;
  carry <= ctr(3);
  process (clk, reset)
  begin
      if reset = '0' then
          ctr <= (others => '0');
      elsif clk'event and clk = '1' then
          if enable = '1' then
              if ctr = "1001" then
                  ctr <= (others => '0');
              else
                  ctr <= ctr + 1;
              end if;
          end if;
      end if;
  end process;
end Behavioral;
```

Between `is` and `begin` is the declarative region where all constants, types, variables, signals, functions, and component declarations must be made. In this case only a single `ctr` signal is declared with a width of 4 bits. The `ctr` represents the internal decade counter state.

Two signal assignments appear outside the decade counter process just after the `begin` keyword of the architecture. In effect, each of these signal assignments is contained in a process of its own with a sensitivity list containing the signal on the right side of the signal assignment.

In terms of hardware, these signal assignments connect wires between the internal counter state and the external pins identified on the left side of the signal assignment. The reason you can't directly use the signal `q` as the state is because its port is defined as a write-only (or out) port. This is a common occurrence with VHDL `out` ports. In fact, some people attempt to cheat and define `out` ports as being `inout` or buffer. Don't do this. Let the data direction be your guide in defining the direction of ports. If the data is truly one-directional, use either the `in` or `out` port. Only bidirectional data flows (e.g., data buses) require `inout` ports.

DECADE COUNTER PROCESS

In the begin/end block is a process block. Processes are used to encapsulate a particular behavior of the design (the decade counter behavior in this case). The `clk` and `reset` arguments following the process name form the process's sensitivity list. This process is activated only when signals in the sensitivity list (`clk` or `reset`) change state. Sensitivity lists are used mainly in optimizing simulations so that process activation can be minimized to speed up the simulation. Synthesis tools monitor the sensitivity lists for completeness; they issue warnings if signals are missing, but they don't require them.

If you don't ensure that the sensitivity lists are complete, you may be surprised with some unusual simulation results. You haven't really used VHDL until this sort of simulation bug bites you.

Within a clocked process, you need to include the clock and all other dependent signals outside the clocked region of code in the sensitivity list. The clocked region in this example is demarcated by the `elsif` and `end if` statements.

The asynchronous part of the decade counter is after the first `if` conditional. The `reset = '0'` condition activates the statements following `then`, which, in this case, clears the `ctr` state.

The <= operator is a signal assignment. The signal assignment is different than the variable assignment. Signals are assigned concurrently; variables are assigned sequentially. This may sound like semantics, but it's actually an extremely important point to master. Signal concurrency is a reflection of the fact that all signal transitions in the real world may occur simultaneously. Consequently, all signals within a VHDL project change state simultaneously.

The strange-looking `(others => '0')` construct is a short-form technique for saying that all the array elements are assigned the 0 value. This

notation is a named association in which elements of an array are assigned by an index number or by the default name others.

You also could explicitly assign array elements with 0 => '0', 1 => '0', 2 => '0', and 3 => '0'. A significant advantage of the others notation is that the array initializer can be independent of an array's size.

Vectors can be initialized using a constant bit vector such as 0000. This approach reduces the amount of writing when initializing bit vectors. Extremely large bit vectors can be abbreviated even more by prefixing x for hexadecimal, o for octal, and b for binary so that 101010000001 can be written as x"A81" or o"5201". The default bit vector radix is assumed to be base two.

The clk'event following the elsif makes use of a signal attribute. Refer to the *Circuit Cellar* FTP site for a list of some common VHDL attributes supported by most synthesis tools. The event attribute triggers the following statements whenever the attached signal changes state. The clk = '1' condition means that the statements within the clock block, from then to end if, become active for every rising edge of the CLOCK signal. Synthesis tools look for this code sequence and produce clocked constructs in such cases. The std_logic_1164 package defines two functions, rising_edge and falling_edge, which duplicate the rising edge statements' functionality. I'll address these functions with examples.

Basic counter functionality is defined by the central if, else, and end if statements. The if enable = '1' then clock enable statement provides a gated clock implementation to apply clock pulses to the counter only when the ENABLE pin is at a logical 1. Then, the counter state is incremented by one (an implicit conversion takes place to convert the one to a std_logic_vector) as long as the counter isn't equal to 1001 or nine, which represents the final decade counter state. When this final count is reached, the counter will perform a synchronous reset on the next rising clock edge.

I know this may seem like a lot of work to get a decade counter, but it's all downhill from here. Armed with these basic concepts, you can design just about any counter or divider.

Figure 2 (page 44) shows the synthesized FPGA structure that results from this VHDL code. The primitives are based on those available in the Xilinx Spartan-3. As you can see, another benefit of VHDL is that you can express a design in a generic language that has no ties to a particular piece of hardware. You can resynthesize this same VHDL code for an Actel FPGA or any other vendor's device, including ASICs! Figure 3 (page 44) is a sample simulated output for approximately 10 clock cycles.

GENERIC ARGUMENTS

Generic arguments are powerful mechanisms available in VHDL for defining general-purpose entities that you can tailor to suit different requirements. For example, Listing 2 contains a generic counter implementation that's easily customized by passing a couple of arguments to transform it into practically every different counter or divider you'll need for my watch project!

Two generic arguments, width and max_cnt, which are located at the top of the entity declaration, are the keys to this flexibility. The former represents the counter's bit width. The default width is specified as eight. The max_cnt argument specifies the maximum count value that the counter can reach. The default max_cnt is 255. Default arguments enable you to create an instance without having to specify the generic arguments. As long as you're happy with the default generic arguments, neither the synthesis tool nor the simulator will complain.

The first place you'll see a change from the original decade counter is in the definition of the q signal. In the first decade counter implementation, I hard-coded an upper limit of three. Here in the port declaration, q's upper limit is set to the value of width-1. Thus, if you were to instantiate this counter with a width of 12, q's range would be 11 down to 0, thus creating a 12-bit counter width.

Listing 2—*The general-purpose synchronous VHDL counter/divider implementation has generic parameters to specify the counter bit width and maximum count value. An asynchronous reset, a clock enable, and a carry output complete the design.*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity divn is
   generic          (width   : integer := 8;
                     max_cnt : integer := 255);
   port             (clk     : in std_logic;
                     reset   : in std_logic;
                     enable  : in std_logic;
                     q       : out std_logic_vector(width-1 downto 0);
                     carry   : out std_logic);
end divn;
architecture behav of divn is
   signal cnt : natural range 0 to max_cnt;
begin
   carry <= enable when cnt = max_cnt else '0';
   q <= conv_std_logic_vector(cnt, width);
   counter: process (clk, reset)
   begin
       if reset = '0' then
            cnt <= 0;
       elsif rising_edge(clk) then
            if enable = '1' then
                if cnt = max_cnt then
                    cnt <= 0;
                else
                    cnt <= cnt + 1;
                end if;
            end if;
       end if;
   end process;
end behav;
```

Another change relates to the data type used for the counter. The original decade counter used an `std_logic_vector` type. For the generic counter, however, I used a subtype of the natural number type for the `cnt` type. The subtype is declared implicitly in the cnt signal declaration as `natural range 0 to max_cnt`. This means that the cnt signal can take on any value from zero to `max_cnt`.

Why did I make this change? It certainly could have used an `std_logic_vector` as a generic argument, but I prefer working in base 10, so it made sense to use a natural number counter. There isn't a hardware penalty for this choice. Plus it makes the design more readable. As a consequence, you need a conversion function (`conv_std_logic_vector`) to convert from the natural number cnt to the `std_logic_vector` result for q. The conversion takes place as part of the signal assignment where the cnt state is wired to the q output. The first argument is the natural or integer signal to be converted to a `std_logic_vector`. The second argument is the desired width of the resultant `std_logic_vector`. This conversion function is built into the IEEE std_logic_arith package.

The remaining code should look extremely similar to the decade counter (with the exception of using natural numbers in places where vector constants were previously required for initialization and maximum count comparison). Another change is the use of the `rising_edge` function to define the clock edge used for event transitions. This makes the code more easily understandable.

The final change has to do with the way you produce the carry output signal. In the decade counter, you simply use the most significant bit as the carry signal. But when creating a chain of synchronous counters, you might sometimes create counters in which the upper bit remains high for longer than one clock cycle. Thus, you could get two or more clocks carrying through to the next counter in the chain.

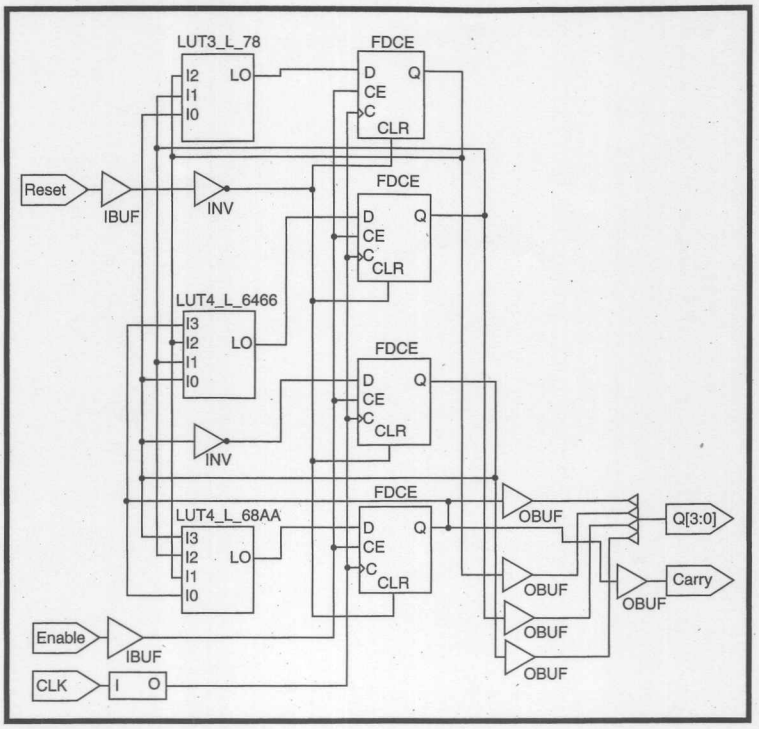


Figure 2—*The Spartan-3 primitive blocks comprise the decade counter. The LUT3 and LUT4 blocks are three- and four-input look-up tables. The FDCE blocks are D flip-flops with clear and enable inputs.*

Because you'll want only one carry clock generated each time the counter reaches its maximum value, that's what I encoded in the signal assignment for the carry output. Note that this is a conditional signal assignment using the `when` conditional statement. The ENABLE signal is passed through to the `carry` whenever the `cnt = max_cnt` condition is satisfied; otherwise, in the `else` clause, a logical 0 value is assigned to the `carry`.

FIRST TEST BENCH

You must create a test bench to test whether or not the generic counter actually works. A test bench is analogous to a hardware bench. It contains instruments such as signal generators, oscilloscopes, and logic analyzers, which are connected to the circuit being tested. The VHDL equivalents of these instruments are shown in Listing 3.

A test bench is nothing more than an entity/architecture pair with the circuit being tested instantiated in the declarative region of the architecture. You can see the generic `divn` component declaration following the `is` of the architecture. Signal declarations for each of the `divn` port elements follow the component. The actual instantiation is found in the architecture body beside the `uut:` label. Each signal passed or received from the `divn` port uses a named association with the port signal name on the left and the local signal on the right of the => symbol.

The primary stimulus required is the clock, which is defined in the process block labeled "clock." Initially, `clk` is set to a logical 0 value.

The following `wait` statement pauses the process for 500 ns. The `wait` statement should never appear in any code that can be synthesized. After the first `wait`, `clk` is set to a logical 1 value and the process waits another 500 ns. After the final `wait` time has elapsed, the process begins executing from the

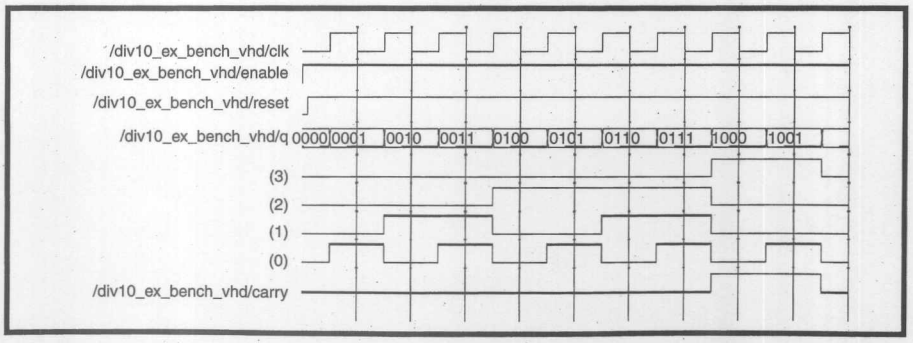


Figure 3—*The top clock waveform drives the simulated decade counter. The q output is shown below in both a vector format and broken down into each of the four counter bits.*

initial statement following the begin statement. In effect, this defined a clock with a 50% duty cycle and a 1-µs period.

The second process, which is labeled tb, provides the required reset stimulus. Like clk, the reset receives an initial value of logical 0. After 100 ns, it's set to logical 1. The ENABLE signal is tied permanently to a logical 1 for the purposes of this test. The final wait statement with no arguments pauses the second process indefinitely. This is an effective method of executing the contents of a process only once. Remember, wait statements can't be synthesized and therefore can't appear in any VHDL intended to be rendered as a hardware component.

As you can see, by defining one or more processes and using wait statements, it should be possible to synthesize any stimulus required for driving a test component's inputs. More elaborate test benches can also include functional component models of elements such as memories, processors, clocks, and data acquisition systems provided by many IC vendors. The simulator running the test bench acts as a logic analyzer and provides access to all the tested model's inputs and outputs.

Listing 3—*The small test bench generates CLOCK and RESET signals to drive the divn counter implementation. The test bench instantiates a decade counter from the divn generic counter.*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity divn_bench_vhd is
end divn_bench_vhd;
architecture behavior of divn_bench_vhd is
--Component declaration for the unit under test (UUT)
   component divn
      generic  (width    : integer;
                max_cnt  : integer);
      port     (clk      : in std_logic;
                reset    : in std_logic;
                enable   : in std_logic;
                q        : out std_logic_vector(width-1 downto 0);
                carry    : out std_logic);
   end component;
--Inputs
   signal clk :  std_logic := '0';
   signal reset :  std_logic := '0';
   signal enable :  std_logic := '0';
--Outputs
   signal q :  std_logic_vector(3 downto 0);
   signal carry :  std_logic;
begin
--Instantiate the unit under test (UUT)
   uut: divn
      generic  map       (width => 4,
                          max_cnt => 9)
      port map           (clk => clk,
                reset => reset,
                          enable => enable,
                          q => q,
                          carry => carry);
   clock : process
   begin
      clk <= '0';
      wait for 500 ns;
      clk <= '1';
      wait for 500 ns;
   end process;
   tb : process
   begin
--Wait 100 ns for global reset to finish
      reset <= '0';
      enable <= '1';
      wait for 100 ns;
      reset <= '1';
      wait; -- will wait forever
   end process;
end;
```

START EXPERIMENTING

Thus far, I've covered a lot of VHDL territory: data types, attributes, entities, architectures, counters/dividers, generic components, test benches, instantiated components, and a variety of statements. Next month I'll investigate state machines, encoders/decoders, multiplexers, and ROMs. I'll show you how these elements can be combined to make the stopwatch application in Figure 1. Until then, download some VHDL tools and have fun experimenting with the examples in this article.

The complete VHDL source code is available on the *Circuit Cellar* FTP site. If you're using the Xilinx tool set, I also included a project that contains all the same source code.

Michael Griebling is a professional electrical engineer with a B.A.Sc. in electrical engineering. He has been active in the aerospace industry for more than 20 years. Michael enjoys designing firmware, hardware, and software. You may reach him at mg@comp-inspirations.com.

PROJECT FILES

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2005/181.

RESOURCES

Std-1076-2002, *IEEE Standard Language Reference Manual*, IEEE, Piscataway, NJ, 2002.

Std-1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, IEEE, Piscataway, NJ, 1993.

Xilinx, *Spartan-3 FPGA Family: Complete Data Sheet*, DS099, January 2005.

——— *Spartan-3 Starter Kit Board User Guide*, UG130, Xilinx, October 2004.

SOURCES

ISE WebPACK 7.1i and XC3S200 Spartan-3 FPGA starter kit
Xilinx
www.xilinx.com

Hardware Synthesis with VHDL (Part 2)

VHDL in Action

Ready to put VHDL in play? Michael shows you how to build state machines, decoders, multiplexers, debounce and edge detection circuits, tristate bus buffers, shift registers, and more.

Last month I described VHDL elements such as packages, entities, architectures, data types, attributes, and generic counters as an introduction to my watch application. Now I'll describe how you can build state machines, decoders, multiplexers, debounce and edge-detection circuits, tristate bus buffers, shift registers, and more.

WATCH BASICS

Let's begin by looking at the Watch component instantiation hierarchy (see Figure 1). The heart of the watch is the Time Base component, which takes a 50-MHz reference frequency from the Spartan-3 starter kit oscillator and divides it down to 1-kHz, 10-Hz, and 1-Hz reference enable clocks using a chain of dividers based on the Divn component. The 1-kHz ENABLE signal drives the display driver LED multiplexing. The 10-Hz ENABLE signal drives the stopwatch and debounce circuitry. The 1-Hz ENABLE signal provides a time base for both the normal date and time and the dual time zone functions, both of which are instantiated by the DateTime component.

The watch's brain resides in the Mode Generation component (see Listing 1, p. 50). The input push buttons and switches are debounced here. It's also where the Set Clock (PB1) push button's rising edge is detected. These inputs are decoded to provide the watch's different modes: the date/time function, the dual time zone, the stopwatch, and the alarm (see Table 1).

The Set Mode push button triggers the watch and alarm setting functions in Table 2. In Set mode, PB1 increments the corresponding watch counter (seconds, minutes, hours, days, months, or years) by one each rising edge pulse.

BOUNCING SWITCHES

The debounce process in Listing 1 conditions the push button and switch inputs. Figure 2a shows a typical switch or push button waveform. Contact bounces occur on each make and break of the switch contacts. This bounce time can range from 5 to 50 ms, depending on the quality of the switch.

The 100-ms enable pulses are used to sample the switch and push buttons once every 0.1 s to filter out the contact bounce. The resultant output is a set of clean switch and push button pulses.

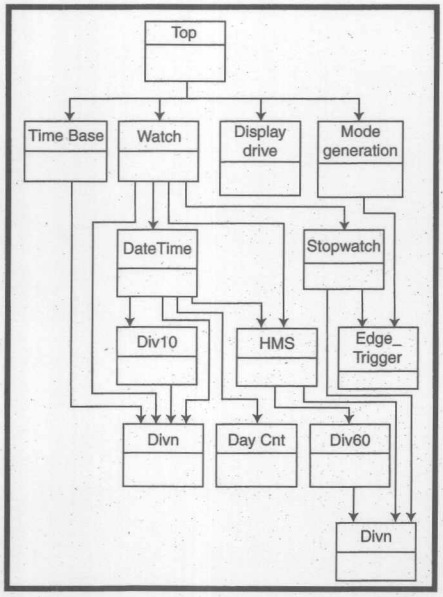


Figure 1—*In the Watch component instantiation hierarchy, the Top component instantiates the Time Base, Watch, Mode Generation, and Display Driver components. The Divn and Edge_Trigger generic components are used throughout the watch application.*

The `en_debounce` signal in Listing 1 occurs every 100 ms. The time base generates it. Both the `pb_in` and `sw_in` signal vectors are debounced and assigned to the `pb` and `sw` vectors, respectively.

DATE/TIME SELECTION

Unfortunately, the Spartan-3 starter kit provides only a four-digit LED display. Thus, to display both the time and date (as most watches do), you need a way to select which output is displayed on the LED.

The `ToggleDateTime` process in Listing 1 accomplishes this by toggling between modes that display either the time or date. PB3 switches between displaying the time or date for both the normal date/time and the dual date/time functions with each activation.

The `ToggleDateTime` process is different from most others because it involves a second clock source (i.e., PB3) to drive the `dtime` signal toggling. Furthermore, this push button is active only for this function while either of the two date/time modes are displayed. Note that only the watch modes change within the Mode Generation component. We'll look at the actual display multiplexing in the Display Driver component.

ON THE EDGE

There are two methods for triggering on a particular edge of an input: either clock a process directly (as I demonstrated with the Date/Time select) or use an edge-detection circuit (see Listing 2, p. 52). An example of how the edge detection works is shown in Figure 2b. The asynchronous input sig-

nal is first clocked to generate a clock-synchronous signal. The synchronous signal is then delayed by one clock cycle.

To decode a rising edge signal, use the equation `x_edge <= x_synced and not x_delayed`. A falling edge is decoded with the equation `x_edge <= not x_synced and x_delayed`.

Like the Divn component, the Edge_Trigger component has a generic input called `rising_edge_pulse`. This component will detect a rising edge if a logical 1 is passed to this port; otherwise, it detects a falling edge.

COMPONENTS

The Watch component interconnects the DateTime, HMS, and Stopwatch components. The HMS instance implements the alarm.

Two instances of the DateTime component become the current date and time and the dual time zone. The HMS instance is used as the central alarm.

The snooze function is implemented in the Watch component. The actual alarm function is contained in the individual DateTime components. The alarm signal is sent back to the Watch component. There are two possible alarm sources from either the current date/time or the dual time zone, so these signals are combined at the Watch level. After the alarm is activated, you have the option of activating a snooze counter (derived from the Divn component) with a 10-min. time out period. A state machine implements the alarm and snooze functions. The alarm reactivates after the snooze counter times out. The alarm can be cleared with SW3 or the snooze counter can be restarted with PB2 for another 10-min. period.

The DateTime component instantiates Divn- and Div10-based components that implement the majority of the date and time counters. This component is also responsible for checking when the alarm matches the current hours and minutes via the `CheckAlarm` process posted on the *Circuit Cellar* FTP site. During synthesis, the = operator produces hardware that compares the alarm signal to the active hour/minute signal. When the two signals are equivalent, the `alarm_on` signal is activated.

The date is probably the trickiest part of the DateTime function. You must automatically adjust the number of days in a month, calculate whether or not the current year is a leap year, and remap both the month and day counters to start from a count of one instead of zero.

The month remapping listing is posted on the *Circuit Cellar* FTP site. A counter range from zero to 11 is mapped to a binary-coded decimal month number that ranges from x01 to x12.

I couldn't use the Divn component for the day counter. Because the maximum day count changes from month to month, the day maximum value must be variable. The day_cnt component implements this function. In addition to the normal arguments of the Divn counter instances, I provided a maximum count value input.

I needed to be innovative with the year counter. First, I used two Div10 (decade) counters to implement the units and tens places in the year. Then the "20" century was hard-coded to expedite the year-setting process. Instead of having to start from a year of zero and increment 2,000 times to reach the right century, you start at a count of 2,000 and increment to the current year. Only years from 2000 to 2099 can be supported. Leap year calculations require a second, 2-bit binary year counter that effectively performs a modulo four operation on the year. The reason this works is that the year 2000 mod 4 equals zero, so both the BCD counters and the binary counter can start from zero. They're guaranteed to synchronize as the year is set.

I haven't implemented a day of the week function mainly because there isn't a nice way to display the result with the Spartan-3 starter kit. Most watches cheat and require you to set the current day of the week along with the date. I would prefer to have this value calculated automatically from the current date. I'll leave this implementation for you to work out.

Also, it would be useful to add a feature that automatically adjusts for daylight saving time. This function would require the correct day of the week, so I omitted it in this watch implementation. Again, it's fairly simple to add, so I'll leave it for you to implement.

The LED display has only four digits. So, how do you display the hours, minutes, and seconds? The solution I came

Function	SW0	SW1	SW3	SW4	PB0	PB1	PB2	PB3
Date/time	0	0	Alarm on/off	12/24 Hours	Toggle Set mode	Set increment	Snooze	Toggle date/time
Second time zone	1	0	Alarm on/off	12/24 Hours	Toggle Set mode	Set increment	Snooze	Toggle date/time
Alarm display	0	1	Alarm on/off	12/24 Hours	Toggle Set mode	Set increment	—	—
Stopwatch	1	1	—	—	Reset	Start/stop	Lap	—

Table 1—*The watch application's modes determine which of four functions are displayed on the four-digit output LED and control the component used during the watch-setting operation.*

Function	mode(6)	mode(5)	mode(4)
Normal timekeeping	0	0	0
Set seconds	0	0	1
Set minutes	0	1	0
Set hours	0	1	1
Set days	1	0	0
Set months	1	0	1
Set years	1	1	0

Table 2—*Seven watch-setting modes give you the ability to preset the seconds, minutes, hours, days, months, and years of the currently selected function. The active Set mode is displayed using discrete output LEDs (LED7, LED6, and LED5.)*

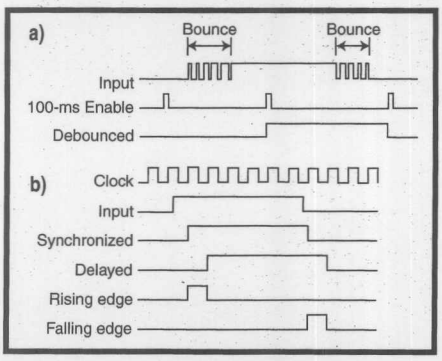
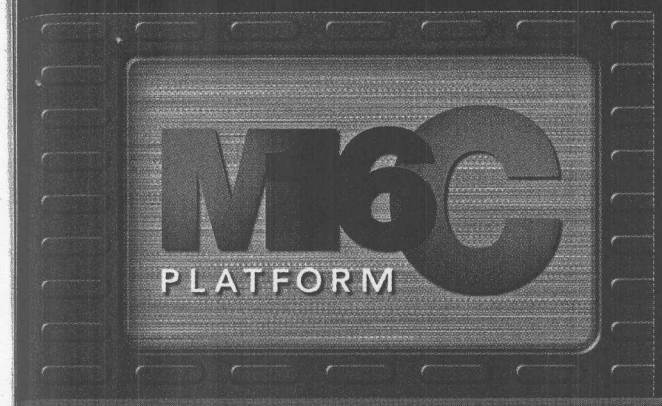


Figure 2a—*The topmost input signal exhibits bounce during switch make and break transitions. By sampling the input signal at 100-ms intervals, a debounced signal is generated.* **b**—*These timing waveforms show how the edge detection is accomplished. By properly decoding a delayed and clock-synchronous signal, you can generate either a rising- or falling-edge pulse.*

up with was to automatically switch between an hours/minutes display and a minutes/seconds display every 5 s. As well, the date display needs to switch between displaying the day/month and the year. A secondary function, when setting the date/time, requires that the correct quantity (seconds, hours, etc.) be displayed. The DriveDisplay process in the DateTime component imple-

Listing 1—*The mode generation component conditions the input push buttons and switches. It also generates the watch modes to control the watch functions.*

```
architecture behav of mode_generation is
--Declarations omitted for clarity
begin
--Generate single rising-edge pulse for set push button
    gen_set_pb: edge_trigger
    port map (
     clk => clk,
     reset => reset,
     x => pb(1),
     x_edge => mode(7)
);
--Inputs are debounced by sampling inputs every 100 ms, thus
--ignoring bounce transitions within this period.
    debounce: process (reset, clk)
    begin
     if reset = '0' then
       pb  <= (others => '0');
       sw  <= (others => '0');
     elsif rising_edge(clk) then
       if en_debounce = '1' then
          pb <= pb_in;
          sw <= sw_in;
       end if;
     end if;
    end process;
--Generate the mode signals
--Omitted for clarity
    GenSetMode: process (set_clk, reset, min_sel)
    begin
     if reset = '0' then
       setmode <= min_sel;
     elsif rising_edge(set_clk) then
       if setmode = max_sel then
          setmode <= min_sel;
       else
          setmode <= setmode + 1;
       end if;
       end if;
    end process;
    GenMinMax: process (sw)
    begin
     case sw(1 downto 0) is
       when "00" | "01" =>
          min_sel <= "000"; --Clock and dual-time can set seconds to years
          max_sel <= "110";
       when "10" =>
          min_sel <= "000"; --Alarm can set seconds to hours
          max_sel <= "011";
       when others =>
          min_sel <= "000"; --Stopwatch can't be set
          max_sel <= "000";
     end case;
    end process;
    ToggleDateTime: process (set_dt, reset)
    begin
     if reset = '0' then
       dtime <= '0';
     elsif rising_edge(set_dt) then
       if tm_en = '1' then
          dtime <= not dtime;
       end if;
     end if;
end process;
end behav;
```



Everywhere you imagine. RENESAS

MCU Solutions Matched To Your Design Imagination

RENESAS IS THE #1 MCU SUPPLIER IN THE WORLD. **www.renesas.com**

M16C Advantages

- Advanced 16-bit and 32-bit CISC cores that perform at up to 32 Dhrystone 1.1 MIPS
- Banked-register architecture optimized for fast context switching
- Wide range of high-performance flash memory: 8 KB to 512 KB
- Best-in-class EMI/EMS noise immunity
- Designed to be pin- and function-compatible between families and packages
- Extensive peripheral integration: everything from POR and LVD to CRC counters

Low-cost Renesas Toolchain

- Fully integrated suite of hardware and software tools
- Highly-optimized C compiler and assembler – FREE (64K limited version): Download latest version from web (www.renesas.com/skp)
- Low-cost Starter Kit (target board, USB debugger, complete toolchain), starting at $49
- Easy online access to application notes, example code, software development tools

Get your product to market faster using Renesas Interactive!

Renesas INTERACTIVE

Evaluate, test, and learn about Renesas products in an interactive online environment.

www.renesasinteractive.com

SKP	Device	C Compiler, Assembler, IDE	Cost
SKP8CMINI13	R5F21134FP	FREE (64K limited version)	$49
SKP8CMINI17	R5F21174FP		$49
SKP16C26A	M30260F8AGP		$49
SKP16C28	M30280FAHP		$49
SKP16C62P	M30626FHPFP		$49
SKP32C83	M30835FJGP		$119
SKP32C84	M30845FJGP		$49

Special pricing is in US dollars from Renesas authorized North American distributors only, and is subject to change.

Popular Devices

Series/ Group	Part Number	CPU (Bits)	Package	Freq. (MHz)	Dhrystone 1.1 MIPS (Max)	Program Flash (KB)	Data Flash (KB) (10K Rewrites)	SRAM (Bytes)	Serial I/O Channels	I^2C	I/O	A/D 10-bit	Ring Osc.	LVD POR	Timers	Price *
R8C/17	R5F21172SP	16	20 Pin SSOP	20	9	8	2	512	1	1	15	4	X	X	3+WDT	$1.90*
R8C/13	R5F21134FP	16	32 Pin LQFP	20	9	16	4	1K	2	–	24	12	X	X	4+WDT	$2.38*
M16C/26A	M30260F8AGP	16	48 Pin LQFP	20	13	64	4	2K	3	1	39	12	X	X	8+WDT	$4.76*
M16C/28	M30281FAHP	16	64 Pin LQFP	20	13	96	4	8K	4	1	55	13	X	X	9+WDT	$5.90*
M16C/62P	M30624FGPGP	16	100 Pin QFP	24	16	256	4	20K	5	3	87	26	X	X	11+WDT	$8.25*
M16C/62P	M30626FHPGP	16	100 Pin QFP	24	16	384	4	31K	5	3	87	26	X	X	11+WDT	$10.05*
M32C/84	M30845FJGP	16	144 Pin LQFP	32	32	512	–	24K	5	5	124	34	X	X	11+WDT	$16.88*

* Suggested 10,000 piece price. Please contact your local Renesas distributor for volume pricing.

Listing 2—*Take a look at the VHDL implementation of an edge-detection function. The first process samples the input signal on the clock's rising edge to provide a clock synchronous signal (x_synced). The x_synced signal is then delayed by one clock cycle and assigned to the x_delayed signal. In the second process, either a rising or falling edge pulse (x_edge) is generated depending on the rising_edge_pulse generic parameter.*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity edge_trigger is
  generic (rising_edge_pulse : std_logic := '1');
  port    (clk     : in std_logic;
           reset   : in std_logic;
           x       : in std_logic;
           x_edge  : out std_logic);
end edge_trigger;
architecture Behavioral of edge_trigger is
  signal x_synced  : std_logic;
  signal x_delayed : std_logic;
begin
  process (clk, reset)
  begin
     if reset = '0' then
       x_synced  <= '0';
       x_delayed <= '0';
     elsif rising_edge(clk) then
       x_synced  <= x;
       x_delayed <= x_synced;
     end if;
  end process;
  process (x_synced, x_delayed)
  begin
     if rising_edge_pulse = '1' then
        x_edge <= x_synced and not x_delayed;
     else
        x_edge <= not x_synced and x_delayed;
     end if;
  end process;
end Behavioral;
```

ments this display switching.

Another complication relates to the 12- and 24-h support implemented in the Hour/Minute/Second (HMS) component. The actual hour counter has a range from zero to 23. In 12-h mode, this range is mapped to binary-coded decimal values from x01 to x12 in two ranges, with the second range marked with a p.m. indicator. In 24-h mode, you end up with binary-coded decimal values from x00 to x23. In effect, the encoder implements a binary-to-binary-coded decimal decoder. The full code is posted on the *Circuit Cellar* FTP site.

STOP TIME

Let's look at the stopwatch in Listing 3 (p. 54) in some detail. It contains a state machine implementation and instantiates an Edge_Trigger component. At the top of the architecture declaration are both the Divn and Edge_Trigger components. Following these components is the first enumerated type declaration called the `sw_state_type`. It consists of three enumerated names (`sw_idle`, `sw_running`, and `sw_freeze`), one

for each of the states of the stopwatch state machine in Figure 3. The state and next_state signal declarations are defined with the sw_state_type. They hold the current and next states for the stopwatch.

Within the body of the architecture, the Edge_Trigger component is instantiated twice, once for each of the push buttons that control the stopwatch. You need to detect the rising edge transition for these signals because you don't want the state machine to oscillate between states while a push button is held in the on state. Unless an edge is used, this is a distinct possibility because the same signal is used as a transition gate to both exit and enter the same state.

The state machine consists of three processes: sync_proc, output_decode, and next_state_decode. The sync_proc process updates the state machine's current state on each rising edge of the clock. The output_decode process generates the stopwatch outputs, which consist of the count_enable signal to enable the clocking of the stopwatch counters at a 10-Hz rate. The external display signal is updated whenever the stopwatch is running or in the idle state. The lap function is implemented by having the stopwatch counters continue to run while in the freeze state even though the actual display remains latched with the previous active stopwatch count. The next_state_decode process determines the value of the next_state signal based on the current state and the start_edge/pause_edge signals. A case statement mirrors the state machine states and if/elsif statements select the values for the next_state signal.

When synthesized, simple state machines are usually implemented as a "one-hot" counter chain. This means that there is one state per register. For state machines with a few states, this is usually an optimal approach because no state decoding is required. State outputs can be generated directly from each of the state registers. Most synthesis tools also give manual control of the state encoding mechanism. Typical options are Gray code, Compact, Sequential, Johnson, One-Hot, and User-defined state encodings. Refer to your favorite tool's documentation for details on how to set these options.

DRIVE THE DISPLAY

The final component in the watch application is the Display Driver component. It decodes the binary-coded decimal display signal into a seven-segment display signal that physically drives the multiplexed LED. Multiplexing drives a single digit of the display at a time so that wiring is minimized. By rapidly sequencing the four digits (at 1 kHz in this case) with the correct information, due to the eye's persistence of vision, the display gives

Listing 3—*The complete VHDL stopwatch implementation contains divn and edge_trigger instances. An internal state machine implementation controls how the counters are incremented and displayed on the four-digit external display. User interaction with the state machine gives the stopwatch's functionality.*

```
architecture behav of stopwatch is
    component divn
--Code removed: See full listing on the Circuit Cellar FTP site
    end component;
    component edge_trigger
    generic (rising_edge_pulse : std_logic := '1');
    port (clk, reset, x : in std_logic; x_edge : out std_logic);
    end component;
--Stop watch state definitions
    type sw_state_type is (sw_idle, sw_running, sw_freeze);
    signal state, next_state : sw_state_type;
--Code removed: See full listing on the Circuit Cellar FTP site
begin
--Generate rising edge pulses for inputs
    gen_start: edge_trigger
    port map (clk => clk, reset => reset,
         x => start_stop, x_edge => start_edge);
  gen_pause: edge_trigger
  port map (clk => clk, reset => reset,
         x => pause, x_edge => pause_edge);
--Code removed: See full listing on the Circuit Cellar FTP site
--Stop watch state machine
  SYNC_PROC: process (clk, reset)
  begin
       if reset = '0' then
          state <= sw_idle;
       elsif rising_edge(clk) then
          state <= next_state;
       end if;
  end process;
--MEALY state machine. Outputs are based on state and inputs
  OUTPUT_DECODE: process (state, enable, stop_watch)
  begin
       if state = sw_idle then count_enable <= '0';
       else count_enable <= enable;
       end if;
       if state = sw_running or state = sw_idle then
          display <= stop_watch;
       end if;
  end process;
  NEXT_STATE_DECODE: process (state, start_edge, pause_edge)
  begin
       next_state <= state;  --default is to stay in current state
       case state is
          when sw_idle =>
             if start_edge = '1' then next_state <= sw_running;
             end if;
          when sw_running =>
             if start_edge = '1' then next_state <= sw_idle;
             elsif pause_edge = '1' then next_state <= sw_freeze;
             end if;
          when sw_freeze =>
             if start_edge = '1' then next_state <= sw_idle;
             elsif pause_edge = '1' then next_state <= sw_running;
             end if;
       end case;
  end process;
  end behav;
```

the appearance of having all four digits on at the same time. A four-digit multiplexed LED requires only 12 connections: four digit drives and eight segment drives (one drives the decimal point). Without multiplexing, the same four-digit LED would require 32 separate connections.

To aid in setting the internal date/time, a flashing function turns on and off pairs of digits in response to the watch's mode. When an alarm is active, all the digits flash. The flashing operation uses a specially encoded segment drive value corresponding to an "1111" input that turns off all the segments. A 1-Hz signal controls the display flashing rate. Refer the *Circuit Cellar* FTP site for the complete VHDL source code for the Display Driver component.

TRISTATE BUFFERS

Although not used in the watch application, it's fairly common to have an FPGA connect to a data bus that's shared with multiple components such as memories, microprocessors, or data acquisition systems. So, how do you implement this with VHDL?

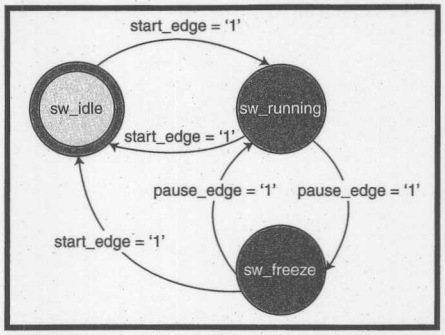


Figure 3—*In the stopwatch state machine, two control inputs (`start_edge` and `pause_edge`) cause transitions among the `sw_idle`, `sw_running`, and `sw_freeze` states.*

The 8-bit tristate buffer posted on the *Circuit Cellar* FTP site reads from and writes to a shared data bus. A READ signal latches the bus contents into an internal register. This is also a typical VHDL latch implementation. A WRITE signal enables the tristate buffer and places the input register's contents on the bus as long as the WRITE signal is active.

Tristate buffers can be implemented only on the external I/O pins of an FPGA. The synthesis tools will automatically "bubble up" tristate pins to the top level of a design so that they are located on external connections.

SHIFT REGISTERS

Implementing a shift register in VHDL is fairly straightforward: Define a vector whose size reflects the number of bits in the desired shift register and use a clocked process to shift the register contents 1 bit at a time. To shift a single bit left, write: `reg <= reg(14 downto 0) & '0'`. Here, the uppermost bit, or `reg(15)` (assuming a 16-bit shift register), is shifted out of the register and a logical 0 is shifted into the least significant bit position.

To reverse the direction of the shift register, write: `reg <= '0' & reg(15 downto 1)`. Some vendors have specific FPGA constructs that efficiently implement shift registers. For example, the Spartan-3 uses an SRL16 component to implement efficient shift registers with up to 16 bits. A popular use of shift registers is to serialize parallel data streams and convert serial communication streams into parallel data.

RAM & ROM

Many FPGAs include memory elements as part of their design. With a proper design, memories can be inferred from the VHDL code by the synthesis engine. For example, to implement an 8 × 8 bit ROM element, start with a type declaration such as `Mem_type = array (0 to 7) of std_logic_vector(7 downto 0)`. Next, a constant with the `Mem_type` needs to be declared along with an initialization constant array: `constant rom : Mem_type := (x"10", x"20", x"30", x"40", x"50", x"60", x"70", x"80")`. For example, the fourth byte of the `rom` constant is accessed as `rom(3)`.

Implementing a RAM is similar, except a signal declaration is used instead of a constant: `signal ram : Mem_type`. No initialization constant array can be used with RAM. Accesses to the RAM can be either synchronous to a clock or asynchronous, depending on the application. The synchronous RAM needs to be contained within a clocked process while the asynchronous RAM can appear in any statement.

MATH FUNCTIONS

Vendors such as Xilinx include hardware multipliers within their FPGA fabric. For example, the XC3S200 FPGA included with the evaluation kit contains 12 18 × 18 bit signed multipliers, each of which produces a 36-bit result. Implementing math functions is fairly easy: just use whichever operator (+, –, *, mod, or /) you require as part of a statement and the synthesis tool will automatically instantiate the appropriate integer math hardware. Of course, indiscriminate use of math operators is guaranteed to quickly chew up FPGA resources, so you may need to make some trade-offs such as sharing math hardware through the use of multiplexers.

BRIDGING CLOCK DOMAINS

In the real world, it is necessary to deal with events that occur on different time scales. For example, in the stopwatch, it was necessary to interface a 10-Hz counter chain with push button controls operated asynchronously by a human. Whenever two or more different clock domains interact, some sort of bridging mechanism is required to ensure that timing glitches can't occur. Recall the edge detection circuit I used to condition the push buttons. The inputs were first synchronized to the clock, and then they were delayed or latched. Latches are a common mechanism to transition between clock domains. Other bridging mechanisms include FIFOs and dual-ported memories.

With a FIFO, data is written in with a first clock, and data is read out with a second clock. Obviously, some synchronization is also required to ensure that the FIFO doesn't overflow and that the reader doesn't run out of data.

A dual-ported memory gives simultaneous access to a shared memory array. Two separately clocked processes can independently access the memory in either Read or Write modes. Of course, some discipline is still required to prevent one process from overwriting data placed in the memory by the second process. Typically, different memory regions are read-only for process one, can be read or write for

process two to prevent write conflicts.

FPGA vendors have FIFO and dual-ported memory solutions specific to their FPGAs. These implementations may make use of special memory arrays in the FPGA fabric, or they may use distributed register-based memory implementations. You'll need to consult your FPGA's documentation to know how to best implement these functions.

VHDL WRAP-UP

In case you're interested, the final watch project used the following resources in an XC3S200-4FT256C FPGA: slice registers (4% used out of 3,840), four-input LUTs (15% used out of 3,840), I/O (19% used out of 173), and global clocks (12% used out of 8). This is the second smallest FPGA in the Spartan-3 family, so you can imagine the amount of functionality that could fit in some of the larger family members with more than 10 times these resources. The maximum estimated speed of this design clocked in at about 76.5 MHz.

Although I've covered a lot of the VHDL language, a great deal remains unsaid. I haven't covered clock synthesis, constraints, and I/O interfacing. The main reason for this is that these topics are extremely dependent on a particular vendor's FPGA implementation. I have also deliberately skimmed over the topics that are required to produce functional VHDL models for simulation. My goal has been to introduce enough VHDL background to provide a solid foundation for hardware synthesis so that a more detailed exploration of VHDL with a particular vendor's toolset is possible.

The cost of using custom FPGAs has been steadily decreasing to a point where it is now possible for hobbyists to include these devices in their designs. VHDL provides a powerful high-level and machine-independent tool to define hardware within these FPGAs. I hope you've found this introduction to VHDL informative and will continue exploring this fascinating subject.

Michael Griebling is a professional electrical engineer with a B.A.Sc. in electrical engineering. He has been active in the aerospace industry for more than 20 years. He enjoys designing firmware, hardware, and software. You may reach him at mg@comp-inspirations.com.

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2005/182.

RESOURCES

Std-1076-2002, *IEEE Standard Language Reference Manual*, IEEE, 2002.

Std-1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, IEEE, 1993.

Xilinx, *Spartan-3 FPGA Family: Complete Data Sheet*, DS099, Jan. 2005.

———, *Spartan-3 Starter Kit Board User Guide*, UG130, Xilinx, Oct. 2004.

SOURCE

ISE WebPACK 7.1i and XC3S200 Spartan-3 FPGA starter kit
Xilinx
www.xilinx.com